Concert Planner

Applicativo Java per il Corpo Musicale
Giuseppe Verdi di Fognano

Manuel Cecere Palazzo

Elaborato di Ingegneria del Software

Corso di Laurea in Ingegneria Informatica
Facoltà di Ingegneria
Università degli studi di Firenze
Settembre 2020

Contents

Chapter 1

Introduzione

1.1 Motivazione e Funzionamento

L'elaborato realizzato prende spunto da una situazione reale osservata nella Banda Musicale Giuseppe Verdi di Fognano, Montale.
L'applicativo gestisce la pianificazione di un concerto, facilitando al maestro del corpo musicale in questione la scelta di quali brani eseguire. Molti dei brani presenti nel repertorio di un corpo musicale richiedono infatti la presenza di strumenti solisti. La situazione si accentua all'interno di gruppi amatoriali, in cui le cosiddette prime parti assegnate sono marcatamente più difficili delle altre, rivolte a musicisti di maggior esperienza, a volte semi professionisti. Dunque il maestro del corpo musicale dovrà sempre tenere di conto quali sono i musicisti presenti e decidere quali brani nel repertorio sono eseguibili in un dato concerto.
L'elaborato in questione si presenta dunque come uno strumento di supporto per il maestro. Esso tiene traccia dei musicisti, il loro ruolo all'interno della banda e la loro disponibilità per il concerto in questione.
L'applicativo elenca anche i brani nel repertorio, ed in particolare evidenzierà di rosso o verde ogni brano a seconda se i requisiti per suonarlo sono soddisfatti. Questi requisiti sono espressi sotto forma di condizioni che esprimono il tipo e numero degli strumenti necessari.
È in oltre possibile aggiungere, modificare, e rimuovere musicisti, segnare se questi sono presenti o assenti. Funzionalità simili sono presenti per i brani, fra cui la possibilità di modificare o aggiungere requisiti per l'eseguibilità di un brano.

1.2 Metodo di realizzazione e strumenti utilizzati

L'applicativo è stato realizzato con il linguaggio Java, attraverso l'uso dell'IDE IntelliJ IDEA.
Esso propone all'utente una **GUI** (graphic user interface) realizzata attraverso **Swing**, un framework per Java, appartenente alle Java Foundation Classes (JFC) e orientato allo sviluppo di interfacce grafiche. Fondamentale nella progettazione dell'interfaccia è stato lo sviluppo di *mock-ups*, inizialmente realizzati su carta e in seguito su MockFlow, sito dedicato alla pianificazione di interfaccie grafiche.
È stato inoltre adottato il pattern **Model-View-Controller**(MVC) dove il model rappresenta la domain logic del nostro programma. In esso è stato anche implementato il pattern comportamentale **Observer**, allo scopo di aggiornare le condizioni di eseguibilità di brani, e il pattern creazionale **Singleton**, utilizzato per il catalogo dei brani e il gestore dei musicisti.
Allo scopo di progettare e illustrare meglio la logica di dominio, contenuta nel package **com.concertPlanner.model**, e le interazioni del modello con la view e il controller, contenuti negli omonimi package, è stato realizzato un class diagram in UML.
Sono state infine realizzati dei test attraverso il framework di unit test JUnit 5. Questi sono organizzati in classi e contenuti nel package **com.concertPlanner.tests**.

Chapter 2

Progettazione

2.1 Use Case

I casi d'uso sono una pratica molto utilizzata nella fase di definizione dei requisiti funzionali.
Un caso d'uso rappresenta un insieme di interazioni tra il sistema e uno o più attori esterni al sistema al fine di raggiungere un obiettivo. Per attore si intende un'entità esterna al sistema che avvia il caso d'uso con uno specifico obiettivo.
Il sistema è descritto in prospettiva black-box/funzionale e dunque le interazioni mostrate non illustrano l'implementazione interna, ma sono principalmente nella forma di stimoli e risposte percepite dall'attore.
Nel diagramma qui illustrato l'unico attore presente è il maestro del corpo musicale, ossia l'utente che utilizza il programma.

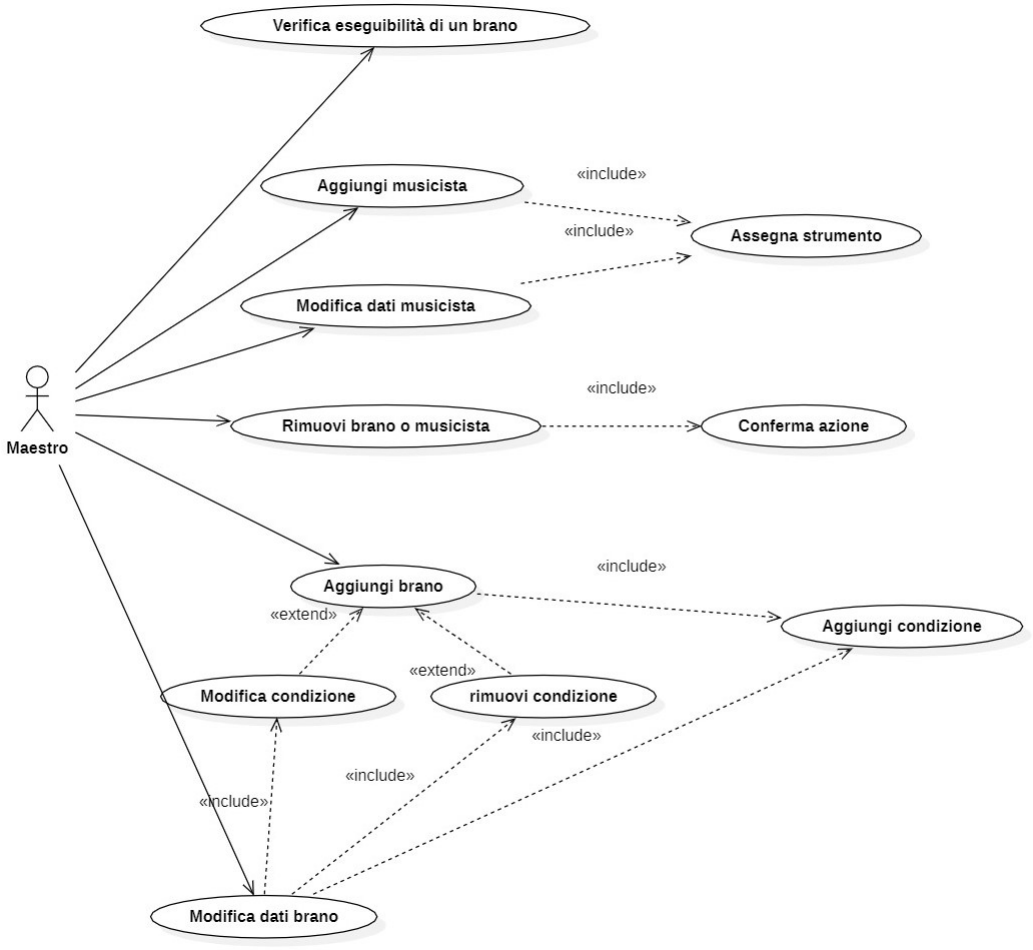

Figure 2.1: Use Case Diagram

2.2 Mock-ups

Un Mock-Up rappresenta un prototipo dell'interfaccia grafica esposta all'utente, senza avere la reale implementazione del programma originale.
Ve ne possono essere diverse realizzazioni nelle varie fasi del ciclo di vita del software, ed è particolarmente utilizzato per presentare al cliente, se presente, un'idea del prodotto finale, allo scopo di ricevere feedback.
Nel caso dell'applicativo in questione, la pratica del mock-up è stata utilizzata in fase di progettazione, realizzando un modello a low-fidelity, allo scopo di valutare le varie idee progettuali e realizzare quali elementi e widget della libreria Swing fossero necessari.
Nonostante i colori, e più in generale il *Look and feel*, siano diversi nel prodotto finale, la struttura degli elementi nelle varie finestre è rimasta inalterata.

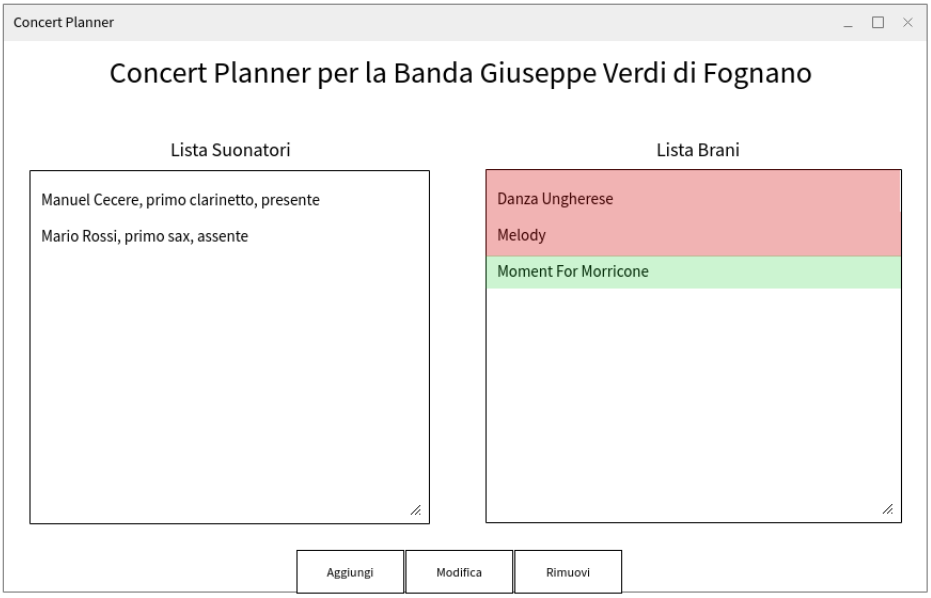

Figure 2.2: Frame principale

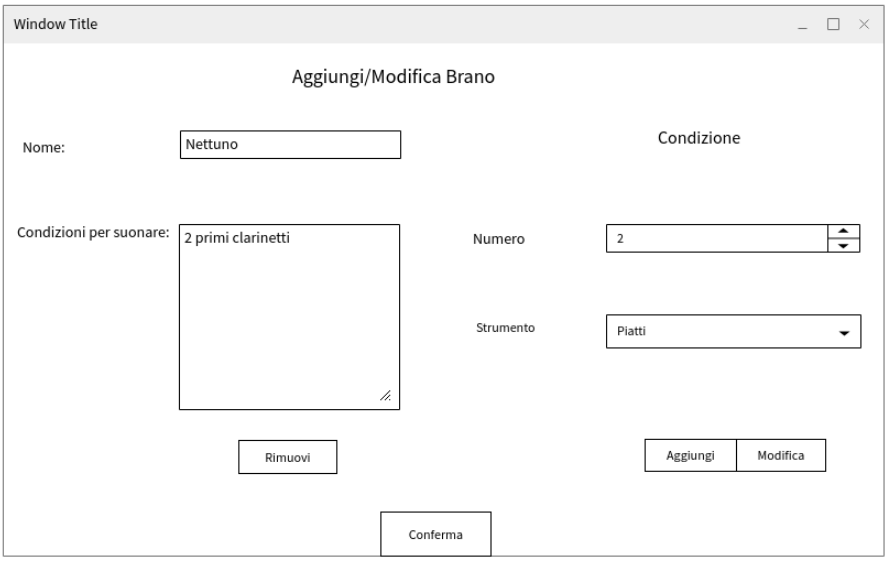

Figure 2.3: Frame di modifica o aggiunta di un brano

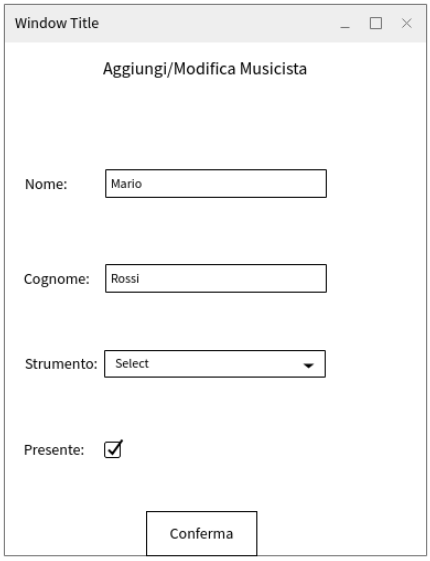

Figure 2.4: Frame di modifica o aggiunta di un musicista

2.3 Class Diagram

Un Class Diagram è un diagramma a struttura statica fondamentale nel modello UML. Esso ha lo scopo di descrivere la struttura del sistema mostrandone le classi, le relazioni fra esse e i loro attributi e metodi. Nei class diagram qui illustrato, si descrive la logica di dominio del programma e la sua collaborazione con le classi dei package che svolgono il ruolo di Controller e View secondo il pattern MVC.

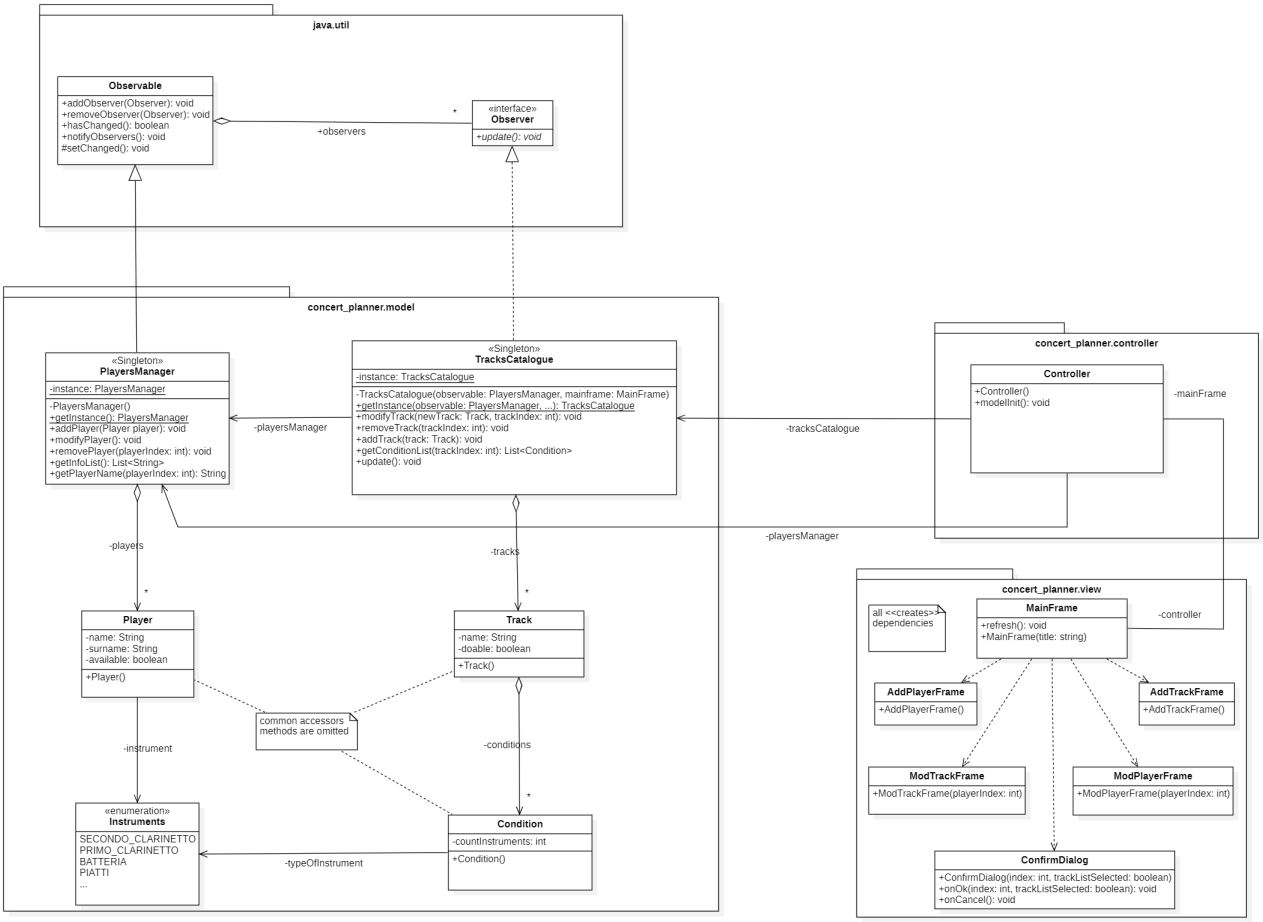

Figure 2.5: Class Diagram

Chapter 3

Struttura del codice

3.1 Introduzione al pattern MVC

La struttura del codice e la definizione dei packages ruota attorno al design pattern Model-View-Controller, comunemente usato nello sviluppo di interfacce utente. Lo scopo di questo pattern è la strutturazione del codice in tre parti, componenti separati ma comunicanti, per facilitarne lo sviluppo indipendente.
Le sue componenti, in precedenza già citate, sono:

- **Model**: indipendente dall'interfaccia, il modello fornisce le funzionalità fondamentali dell'applicazione e gestisce i dati di essa.
- **Controller**: riceve i comandi dell'utente e li traduce in richieste per il model o la view. Funge quindi da ponte tra i due elementi.
- **View**: visualizza i dati del model all'utente e permette ad esso di interagire con il controller, fornendo i necessari elementi grafici come bottoni o campi di testo.

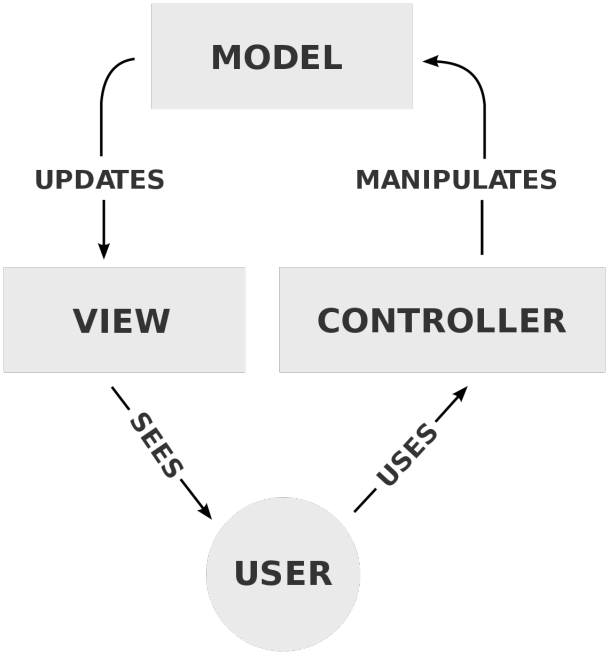

Figure 3.1: MVC design pattern, la figura 2.5 mostra l'implementazione di questo pattern nella struttura delle classi

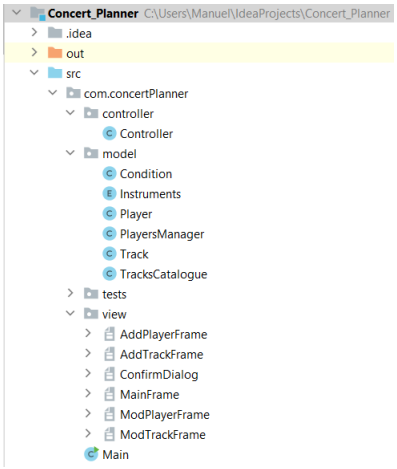

Figure 3.2: Packages e classi

3.2 Model

3.2.1 Instruments

Per definire meglio gli strumenti, utilizzati dai musicisti e presenti nei requisiti per suonare un brano, è stato scelto di realizzare un'**enumeration**, data la natura costante degli strumenti, che non hanno necessità di essere cambiati nel tempo e facilita estensioni future.
Attualmente l'enum *Instruments* include più di dieci elementi, distinguendo fra primi e secondi strumenti, riferendosi con questo termine agli spartiti che vengono assegnati.
Ponendo l'esempio di un *Primo clarinetto* e di un *Secondo clarinetto*, la differenza fra questi all'interno di un corpo musicale amatoriale sarà nella difficoltà dello spartito ed è logico pensare che nelle condizioni per suonare un brano apparirà più probabilmente la presenza necessaria di un Primo clarinetto.

3.2.2 Player

Player è la classe realizzata per rappresentare un musicista.
Le informazioni che questa classe contiene sono il nome e cognome del suonatore, lo strumento (appartenente all'enum *Instruments*) e se questo sarà presente o meno al concerto.
Ognuna di queste informazioni è modificabile dall'utente.

3.2.3 Condition

La classe **Condition** è stata realizzata per rappresentare al meglio ogni singolo requisito che un brano può avere.
Condition presenta il tipo di strumento necessario e quanti elementi di questo devono essere presenti per avere la condizione soddisfatta.
Nella finestra di modifica del brano è possibile aggiungere, rimuovere o modificare queste condizioni.

3.2.4 Track

Un singolo brano è rappresentato dalla classe **Track**. Questa presenta un nome, una lista di condizioni da soddisfare affinchè l'attributo booleano *doable* abbia valore *true*.
Ogni brano è modificabile dall'utente, ma per ovvie ragioni l'attributo doable non può essere modificato direttamente dall'interfaccia, ma dipende dai musicisti presenti ed è aggiornato ad ogni modifica.

3.2.5 PlayersManager e TracksCatalogue

Le classi **PlayersManager** e **TracksCatalogue** sono elementi chiave per il programma sia a livello strutturale che funzionale. Prima di vederle più nel dettaglio, illustriamo i design pattern che queste classi implementano.

Singleton

Il pattern **Singleton** è un pattern creazionale, che ha lo scopo di assicurare che una classe abbia solo un'istanza e fornire un punto globale di accesso ad essa. Per fare ciò realizziamo un metodo statico *getInstance()* che ritorni l'istanza unica della classe se già creata, altrimente la crea e la fornisce. Per assicurarsi che il client non chiami il costruttore della classe, questo può essere dichiarato privato. Sia la classe *TracksCatalogue* che *PlayersManager* sono Singleton. Qui sotto possiamo vedere un frammento di codice di quest'ultima.

```
6
7   public class PlayersManager extends Observable {
8       private List<Player> players;
9       private static PlayersManager instance= null;
10
11      //Item 3
12      private PlayersManager() {
13          this.players = new ArrayList<>();
14      }
15
16      public static PlayersManager getInstance(){
17              if(instance==null){
18                  instance = new PlayersManager();
19              }
20              return instance;
21      }
```

Figure 3.3: Frammento di codice di PlayersManager

Observer

Il pattern **Observer** è un pattern comportamentale, il cui scopo è definire una dipendenza uno-a-molti affinchè, al variare dell'uno, gli osservatori siano notificati di ciò e possano cambiare lo stato se necessario. Dunque avremo un elemento con il ruolo di *Subject*, il cui compito sarà notificare del proprio cambiamento tutti coloro che lo stanno osservano, denominati *Observer* e salvati in una lista, in modo che il Subject possa tenerne traccia. Per l'implementazione sono state utilizzate la classe Java **Observable** e l'interfaccia **Observer**, presenti nel package **java.util**. Nel ruolo di Subject troviamo il *PlayersManager*, che notifica il *TracksCatalogue* (nel ruolo dell'Observer) di ogni modifica fatta alla lista dei musicisti, in modo da verificare l'eseguibilità dei brani.

```
23      public void addPlayer(Player player){
24          players.add(player);
25          setChanged();
26          notifyObservers();
27      }
```

Figure 3.4: Frammento di codice che illustra un esempio di quando viene chiamato il metodo notify

```
83      @Override
84      public void update(Observable o, Object arg) {
85          for (Track track : tracks) {
86              track.setDoable(true);
87              for (Condition condition : track.getConditions()) {
88                  int instrumentCount = condition.getCountInstruments();
89                  for (int i = 0; i < playersManager.getPlayersLenght(); i++) {
90                      if (condition.getTypeOfInstrument() == playersManager.getPlayerInstrument(i)
91                              && playersManager.isPlayerAvailable(i)) {
92                          instrumentCount--;
93                      }
94                  }
95                  if (instrumentCount > 0) {
96                      track.setDoable(false);
97                  }
98              }
99          }
100
101     }
102
```

Figure 3.5: Frammento di codice che illustra l'update dell'Observer

Dunque il compito di PlayersManager è tenere traccia dei musicisti attraverso la lista *players*, e notificare TracksCatalogue ad ogni modifica, aggiunta o rimozione. A causa di ciò esso non potrà mai fornire i riferimenti della lista al Controller con cui comunica, o in generale a qualsiasi altra classe, perchè in quel caso si rischierebbe una modifica senza notificare l'Observer e dunque senza aggiornamento dell'eseguibilità dei brani.
PlayersManager dunque presenta un insieme di metodi che forniscono o modificano dati sui musicisti, usando un indice numerico per selezionare l'elemento scelto dall'utente.
Non solo, anche TracksCatalogue presenta metodi simili, seppure per motivi più legati al pattern Model-View-Controller. Fornire infatti un riferimento ai dati lasciando libertà alle classi della View di modificarli senza interfacciarsi con il Model violerebbe il disaccoppiamento introdotto dal pattern MVC.

3.3 View

Per realizzare la View è stato utilizzato il framework **Java Swing**, usando come strumento di supporto il form builder di IntelliJ IDEA, allo scopo di facilitare la realizzazione e personalizzazione dei Frame.
Ogni classe del package **com.concertPlanner.view** rappresenta una finestra che si può aprire durante l'interazione dell'utente con il programma, e ciascuna di esse estende la classe JFrame, con l'eccezione della classe **ConfirmDialog** che estende la classe JDialog, generata nel caso di una rimozione di un elemento.
Ciascuna di queste possiede un riferimento alla classe *Controller*.
La risposta degli elementi interagibili dell'interfaccia è ottenuta tramite la definizione di *listeners* che invocano il metodo desiderato.
Ogni comportamento non legale, come il mancato inserimento di valori necessari alla creazione di un musicista o di un brano, genera un messaggio di errore tramite il metodo *showMessageDialog* della classe **JOptionPane**.

3.3.1 MainFrame

La classe **MainFrame** rappresenta la finestra principale del programma.
Questa elenca i suonatori e i brani, evidenziando di rosso o verde il brano a seconda se questo sia eseguibile o meno.
Fornisce una toolbar per la modifica, rimozione o aggiunta di brani o musicisti, aggiornando dinamicamente il comportamento dei bottoni a seconda dell'ultimo elemento selezionato.
Ha il compito di generare tutte le altre finestre, passando al momento della creazione il riferimento del Controller.
Ad ogni modifica del Model, il Controller richiama un metodo di aggiornamento per ridisegnare gli elementi delle liste dei suonatori e dei brani della MainFrame, implementando di fatto un pattern Observer.

```
102     public void refresh() {
103
104         playersList.setListData(controller.getPlayerInfos());
105         tracksList.setListData(controller.getTrackNames());
106         tracksList.setCellRenderer(new DefaultListCellRenderer() {
107
108             @Override
109             public Component getListCellRendererComponent(JList list, Object value, int index,
110                                                           boolean isSelected, boolean cellHasFocus) {
111                 Component c = super.getListCellRendererComponent(list, value, index, isSelected, cellHasFocus);
112                 if (controller.istrackDoable(index)){c.setBackground(new Color(0x9566E034, true));}
113                 else {c.setBackground(new Color(0x8BB60F0F, true));}
114                 return c;
115             }
116             });
117
118     }
119
```

Figure 3.6: Frammento di codice che mostra il metodo di aggiornamento del MainFrame

3.3.2 Altri frames

I frame di modifica e aggiunta di musicisti e brani non presentano caratteristiche particolari e la loro struttura è illustrata nelle figure 2.3 e 2.4. Distinzione principale tra le finestre di modifica e le finestre di aggiunta sta nella presenza o meno del caricamento iniziale dei dati dell'elemento selezionato. La finestra di rimozione si presenta come un messaggio di conferma dell'azione, ed è rappresentata da un'unica classe, indifferentemente dalla scelta di un brano o di un musicista.

3.4 Controller

La classe **Controller** corrisponde per ruolo al Controller nel pattern MVC. I suoi metodi sono innescati dai *listeners* degli elementi interagibili della View, e sono richieste di accesso o modifica dei dati che il Model gestisce. Oltre a questo ha il compito di inizializzare il *PlayersManager* e il *TracksCatalogue*, mantenendo dei riferimenti ad essi per la comunicazione con il Model.

3.5 Sequence Diagram

Per illustrare meglio le interazioni tra le varie componenti è qui illustrato un Sequence Diagram che simula il flusso di esecuzione del programma in caso di inserimento di un nuovo musicista. Il ruolo di client in questo diagramma è svolto dalla View.

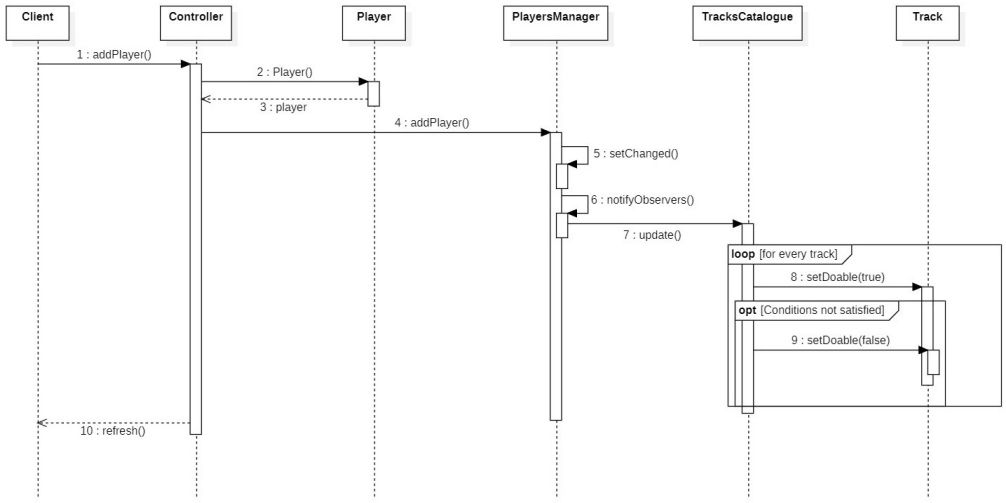

Figure 3.7: Sequence Diagram

Chapter 4

Unit Testing

4.1 Introduzione a JUnit

Lo **Unit Testing** è un metodo di testing del software in cui vengono individuate e testate porzioni del codice per verificarne il corretto funzionamento. Nell'ambito della programmazione ad oggetti (o **OOP**) queste porzioni di codice sono tipicamente classi o singoli metodi.
Nel nostro caso sono stati testati le classi del package
com.concertPlanner.model, allo scopo di assicurarsi il corretto funzionamento dei pattern Observer e Singleton, oltre ai vari metodi di classi chiavi come PlayersManager e TracksCatalogue. I vari **test-cases** sono stati quindi raccolti in classi corrispondenti alle classi del Model.

Nel progetto è stato utilizzato **JUnit 5**, un framework di unit testing. Con JUnit i metodi di test sono sempre preceduti dall'annotazione **@Test**. Se la situazione lo richiede è possibile definire metodi che devono essere eseguiti prima o dopo ciascun test, o prima o dopo tutti i test, attraverso annotazioni come *@Before* (o *@After*) e *@BeforeClass* (o *@AfterClass*). È il caso del metodo **setUp** presente in ogni classe di testing; questo metodo inizializza gli elementi necessari per i test, e viene eseguito prima di ognuno di essi tramite l'annotazione *BeforeEach*.
JUnit fornisce metodi per verificare asserzioni elementari nei vari test cases, alcuni di questi sono **assertEquals, assertTrue, assertFalse** ed altri ancora.

4.2 PlayerTest

La classe **PlayerTest** è stata una delle prime classi realizzate, allo scopo di prendere la mano con la struttura di JUnit. I metodi qui testati sono quelli di lettura e scrittura dei dati.

```
class PlayerTest {
    private Player player;
    @BeforeEach
    void setUp() {
        player=new Player( name: "Manuel", surname: "Cecere", Instruments.SECONDO_CLARINETTO, available: true);
    }

    @Test
    void getName() {
        assertEquals( expected: "Manuel",player.getName());
    }

    @Test
    void setName() {
        player.setName("Marco");
        assertEquals( expected: "Marco",player.getName());
    }
```

Figure 4.1: Frammento di codice della classe PlayerTest

4.3 ConditionTest

La classe **ConditionTest** è stata realizzata con le stesse motivazioni di PlayerTest. I metodi testati sono quelli di letturra e scrittura dei dati.

4.4 TrackTest

La classe **Track** presenta una lista di Condition, dunque per testare i metodi *getCondition* e *setCondition* usando il metodo *AssertEqual* è stato necessario passare esattamente la stessa lista di condizioni di cui poi si testava l'uguaglianza, mantenendo quest'ultima come attributo della classe **TrackTest**.

```java
class TrackTest {
    private Track track;
    private List<Condition> conditions;

    @BeforeEach
    void setUp() {
        conditions = new ArrayList<>();
        conditions.add(new Condition( countInstruments: 2, Instruments.CASSA));
        conditions.add(new Condition( countInstruments: 1,Instruments.BATTERIA));
        track= new Track( name: "Omaggio a Morricone",new ArrayList<>(conditions));
    }
    @Test
    void getConditions() { assertEquals(conditions,track.getConditions()); }

    @Test
    void getName() { assertEquals( expected: "Omaggio a Morricone",track.getName()); }
```

Figure 4.2: Frammento di codice della classe TrackTest

4.5 PlayersManagerTest

Nella classe **PlayersManagerTest** è stato testato il corretto funzionamento del pattern Singleton per la classe **PlayersManager**, controllando che il metodo *getInstance* funzionasse correttamente.
Inoltre data la sua caratteristica di Singleton è stato necessario implementare un nuovo metodo *clearAll* utilizzato in *setUp* per evitare che qualsiasi modifica fatta alla lista dei Player in test precedenti alterasse i risultati dei test successivi.

```java
class PlayersManagerTest {
    private PlayersManager playersManager;
    @BeforeEach
    void setUp() {
        playersManager= PlayersManager.getInstance();
        playersManager.clearAll();
        playersManager.addPlayer(new Player( name: "Manuel", surname: "Cecere Palazzo",Instruments.SECONDO_CLARINETTO, available: true));
        playersManager.addPlayer(new Player( name: "Mario", surname: "Rossi",Instruments.SECONDO_SAX, available: true));
    }

    @Test
    void getInstance() {
        assertEquals(playersManager,PlayersManager.getInstance());
    }

    @Test
    void getInfoList() {
        List<String> strings = new ArrayList<>();
        strings.add("Manuel Cecere Palazzo, secondo_clarinetto, presente");
        strings.add("Mario Rossi, secondo_sax, presente");
        assertEquals(strings, playersManager.getInfoList());
    }
```

Figure 4.3: Frammento di codice della classe PlayersManagerTest

4.6 TracksCatalogueTest

Per la classe **TracksCatalogueTest** vale lo stesso ragionamento fatto per PlayersManagerTest, e dunque anche qui è stato introdotto un metodo *clearAll*. Oltre a testare il pattern Singleton è stato testato anche il pattern Observer, controllando che i metodi *notifyObservers* e *update* fossero invocati correttamente.

```
class TracksCatalogueTest {
    private TracksCatalogue tracksCatalogue;
    private PlayersManager playersManager;

    @BeforeEach
    void setUp() {...}

    @Test
    void getInstance() { assertEquals(tracksCatalogue,TracksCatalogue.getInstance(playersManager)); }

    @Test
    void update() {
        playersManager.removePlayer( playerIndex: 0);
        assertFalse(tracksCatalogue.isTrackDoable( trackIndex: 0));
    }

    @Test
    void getNamesList() {
        List<String> strings = new ArrayList<>();
        strings.add("Nettuno");
        strings.add("Memory");
        strings.add("Danza Ungherese");
        assertEquals(strings,tracksCatalogue.getNamesList());

    }
```

Figure 4.4: Frammento di codice della classe TracksCatalogueTest